MP24 @ II UWr 6 czerwca 2024 r.

Zadanie nr 3 na pracownię

Zadanie na trzecią pracownię polega na implementacji prostego kompilatora języka imperatywnego IMP z wykładu rozszerzonego o możliwość definiowania funkcji. Język będzie kompilowany na prostą maszynę wirtualną opisaną w następnej sekcji. Ta maszyna wirtualna różni się od tej z drugiej tercji wykładu przede wszystkim tym, że nie ma wyróżnionego środowiska — wszystko jest trzymane na stosie i odpowiednie zarządzanie stosem stanowi główną trudność tego zadania.

Maszyna wirtualna

Maszyna wirtualna wykonuje ciąg instrukcji używając przy tym stosu wartości i specjalnej komórki pamięci (rejestru), którą będziemy nazywać akumulatorem. Maszyna została zaprojektowana specjalnie dla języka IMP i posiada cechy przydatne to implementacji tego języka. W szczególności wszystkie wartości na stosie są mutowalne.

Za formalną specyfikację maszyny przyjmujemy implementację maszyny, którą można znaleźć w module `EvalSpec` szablonu rozwiązania (moduł ten korzysta z funkcji pomocniczych z modułu `InfLib`). Ta implementacja stara się wyłapać wszystkie niepoprawne zachowania programu, będąc przy tym czytelna jak się da. Poniżej znajduje się nieformalny opis instrukcji maszyny.

`TOP` Zapisuje do akumulatora adres pierwszej wartości na stosie. Instrukcja działa poprawnie, gdy stos jest pusty, ale zapisana wartość nie ma żadnego sensu.

`LEA` *`n`* Oczekuje, że w akumulatorze znajduje się poprawny adres komórki na stosie (otrzymany przy pomocy instrukcji `TOP` albo `LEA`). Instrukcja modyfikuje ten adres tak, by wskazywał komórkę o n wcześniejszą. Liczba n musi być nieujemna i nie może przesunąć wskaźnika dalej niż do pierwszego adresu poza stosem.

ENTER *n* Odkłada n wartości na stos (n musi być nieujemne). Odłożone wartości są nieokreślone.

LEAVE *n* Zdejmuje i wyrzuca n wartości ze stosu (n musi być nieujemne). Wartość w akumulatorze pozostaje bez zmian.

PUSH Odkłada wartość zapisaną w akumulatorze na stos. Wartość w akumulatorze pozostaje bez zmian.

LOAD *n* Oczekuje, że w akumulatorze znajduje się poprawny adres komórki na stosie. Ładuje do akumulatora wartość komórki o n wcześniejszej. Liczba n musi być nieujemna.

STORE *n* Oczekuje, że pierwsza wartość na stosie jest poprawnym adresem. Instrukcja zdejmuje ten adres ze stosu, a następnie zapisuje wartość akumulatora pod adres o n wcześniejszy. Liczb n musi być nieujemna.

CONST *n* Zapisuje do akumulatora liczbę n.

PRIM *op* Zdejmuje ze stosu wartość która powinna być liczbą. Następnie wykonuje operację arytmetyczną *op* na tej liczbie i wartości w akumulatorze, która też powinna być liczbą. Wynik zostaje zapisany w akumulatorze.

CMP *op* Podobne do instrukcji `PRIM`, ale używa operacji porównania. Zapisuje wartość 1 do akumulatora, jeśli liczby są w podanej relacji, oraz wartość 0 w przeciwnym wypadku.

CALL *f* Wołanie funkcji f. Na stosie jest odkładany adres powrotu (lista pozostałych instrukcji do wykonania), a następnie maszyna zaczyna wykonywać instrukcje funkcji f.

RET Powrót z funkcji. Maszyna zdejmuje ze stosu wartość, która powinna być poprawnym adresem powrotu wygenerowanym przez instrukcję `CALL`. Maszyna porzuca bieżący ciąg instrukcji i zaczyna wykonywać ciąg instrukcji z przeczytanego adresu powrotu. Wartość w akumulatorze pozostaje bez zmian.

READ Instrukcja odczytuje liczbę ze standardowego wejścia i umieszcza ją w akumulatorze.

`WRITE` Instrukcja wyświetla wartość z akumulatora. Wartość ta musi być liczbą.

`BRANCH(`*`cs1`*`,`*`cs2`*`)` Instrukcja oczekuje, że wartość w akumulatorze jest liczbą. Jeśli ta liczba jest równa 0, to maszyna wykona ciąg instrukcji *cs2* zanim przejdzie do następnej instrukcji. Jeśli wartość akumulatora jest niezerowa, maszyna wykona ciąg instrukcji *cs1*, a następnie przejdzie do kolejnej instrukcji.

`WHILE(`*`cs1`*`,`*`cs2`*`)` Maszyna wykonuje ciąg instrukcji *cs1*. Po wykonaniu tego ciągu, w akumulatorze powinna znaleźć się liczba. Jeśli ta liczba jest niezerowa, maszyna wykona ciąg instrukcji *cs2*, po którym ponowi wykonanie instrukcji `WHILE`. W przeciwnym razie maszyna przechodzi do kolejnej instrukcji.

Wszelkie wątpliwości wynikające z tego opisu należy wyjaśniać, zaglądając do formalnej specyfikacji, czyli pliku `src/evalSpec.ml` znajdującego się w szablonie rozwiązania.

Język źródłowy

Zadanie polega na uzupełnieniu implementacji kompilatora znajdującego się w pliku `src/solution.ml`. Zgłaszając swoje rozwiązanie, należy przesłać tylko ten plik, ale oczywiście na potrzeby testowania można modyfikować inne pliki.

Napisanie kompilatora może na początku wydawać się karkołomnym zadaniem. Jednak należy pamiętać, że zajmujemy się tylko zagadnieniem tłumaczenia języka do i tak dość wysokopoziomowej maszyny wirtualnej, która ma instrukcje strukturalne takie jak bloki warunkowe oraz pętle. W dodatku nie należy przejmować się wydajnością generowanego kodu (w prawdziwych kompilatorach, tym zajmują się późniejsze fazy kompilacji). Aby jeszcze ułatwić to zadanie, sugerujemy rozbudowywanie kompilatora inkrementacyjnie, dodając obsługę kolejnych cech języka.

Poniżej znajduje się nasza propozycja podziału zadania na mniejsze podzadania. Za rozwiązanie tylko części z nich można dostać niezerową liczbę punktów. Ponadto, uruchamiając szablon z flagą `-sublang` można dodatkowo wymusić sprawdzanie, czy program wejściowy mieści się w podanym podjęzyku.

Wyrażenia

(Opcja `-sublang expr`)

Na początek możesz założyć, że program składa się tylko z:

- (potencjalnie pustego) ciągu deklaracji zmiennych;
- pojedynczego bloku instrukcji, złożonego z:
  - ciągu instrukcji `read` x, wczytującego wszystkie zadeklarowane zmienne w kolejności ich deklaracji;
  - pojedynczej instrukcji `write` e. Oczywiście, w wyrażeniu e nie może występować wywołanie funkcji.

Na przykład poniższy program mieści się w tym podjęzyku.

```
var x
var y
begin
  read x
  read y
  write x * x + y
end
```

Wynik kompilacji (wydrukowany przez szablon uruchomiony z flagą `-dump`) może wyglądać następująco.

```
 0: READ
 1: PUSH
 2: READ
 3: PUSH
 4: TOP
 5: LOAD  1
 6: PUSH
 7: TOP
 8: LOAD  2
 9: PRIM  MUL
10: PUSH
11: TOP
12: LOAD  1
13: PRIM  ADD
14: WRITE
```

Oczywiście nie jest to jedyne możliwe rozwiązanie. Inne poprawne rozwiązanie może wyglądać następująco.

```
 0: ENTER 2
 1: TOP
 2: PUSH
 3: READ
 4: STORE 0
 5: TOP
 6: PUSH
 7: READ
 8: STORE 1
 9: TOP
10: LOAD  0
11: PUSH
12: TOP
13: LOAD  1
14: PRIM  MUL
15: PUSH
16: TOP
17: LOAD  2
18: PRIM  ADD
19: WRITE
```

Instrukcje

(Opcja `-sublang stmt`)

W dalszej kolejności można zaimplementować pełny język IMP z wykładu. Można zignorować deklaracje i wywołania funkcji oraz instrukcję `return`. Dla przykładu, wynik kompilacji programu

```
var x
begin
  read x
  if x = 0 then skip
  else while true do write 42
end
```

mógłby wyglądać następująco.

```
 0: ENTER 1
 1: TOP
 2: PUSH
```

```
 3: READ
 4: STORE 0
 5: TOP
 6: LOAD  0
 7: PUSH
 8: CONST 0
 9: CMP   EQ
10: BRANCH
    ELSE
12:   WHILE
12:     CONST 1
      DO
14:     CONST 42
15:     WRITE
      DONE
    END
```

Funkcje

(Opcja `-sublang func`)

Następnie rozszerzamy język o funkcje. Teraz program składa się ciągu deklaracji zmiennych, ciągu definicji funkcji oraz instrukcji do wykonania. Poniżej znajduje się przykładowy program korzystający z funkcji.

```
var x

function foo(x, y)
  var z
begin
  z := x * x
  return z + y
end

begin
  x := foo(13,42)
end
```

Każda funkcja składa się z nazwy (`foo`), potencjalnie pustego ciągu parametrów formalnych (`x` oraz `y`), ciągu deklaracji zmiennych (`z`) oraz instrukcji będącej ciałem funkcji (w dalszej części pozwolimy na zagnieżdżone funkcje, ale teraz zignoruj ten fakt). Możesz założyć, że funkcje nie mogą korzystać ze zmiennych

globalnych, ale mogą być wzajemnie rekurencyjne. Dodatkowo, każda ścieżka wykonania funkcji musi przechodzić przez instrukcję `return`, która przerywa wywołanie funkcji i zwraca wartość podanego wyrażenia. Instrukcja `return` nie może znajdować się w ciele samego programu. Przykładowy wynik kompilacji powyższego programu może wyglądać następująco.

```
 0: ENTER 1
 1: TOP
 2: PUSH
 3: CONST 13
 4: PUSH
 5: CONST 42
 6: PUSH
 7: CALL  foo
 8: LEAVE 2
 9: STORE 0
  foo:
11: ENTER 1
12: TOP
13: PUSH
14: TOP
15: LOAD  4
16: PUSH
17: TOP
18: LOAD  5
19: PRIM  MUL
20: STORE 0
21: TOP
22: LOAD  0
23: PUSH
24: TOP
25: LOAD  3
26: PRIM  ADD
27: LEAVE 1
28: RET
```

Zwróć uwagę, jak przekazywane są parametry do funkcji: są one odkładane na stos tuż przed wywołaniem (instrukcje nr 4 i 6) i zdejmowane po powrocie z funkcji (instrukcja nr 8). Dzięki temu funkcja może mieć dostęp do swoich argumentów. W instrukcji nr 15 ładowana jest wartość argumentu `x`. Przed nim jednak znajdują się cztery inne rzeczy: adres zmiennej `z`, zmienna `z`, adres powrotu oraz zmienna `y`.

Gdy dołożyliśmy do języka funkcje, to kolejność obliczeń wewnątrz wyrażeń zaczyna mieć znaczenie. Przyjmujemy, że wyrażenia są obliczane od lewej do prawej, a operatory logiczne `and` i `or` są leniwe: nie liczą drugiego operandu, gdy wynik znany jest już po obliczeniu pierwszego. Dla przykładu program

```
function trace(x)
begin
  write x
  return x
end

function add(x, y)
return x + y

begin
  if add(trace(1), trace(2)) = trace(3) or trace(4) + 1 = 5 then
    write 6
  else
    skip
end
```

powinien wyświetlić następujące liczby.

```
1
2
3
6
```

Zagnieżdżone funkcje (uproszczone)

(Opcja `-sublang nested`)

W tym wariancie pozwolimy na zagnieżdżone funkcje, ale wciąż nie pozwolimy im korzystać ze zmiennych innych niż argumenty formalne i zmienne lokalne. Mogą być natomiast wzajemnie rekurencyjne: funkcja może wołać swoje dzieci, siebie, swoich braci oraz swoich przodków w hierarchii i ich braci. To zadanie jest tylko odrobinę trudniejsze od poprzedniego: wystarczy spłaszczyć hierarchię i przenazwać funkcje tak, by ich nazwy nie kolidowały. Na przykład, program

```
function foo()
  function bar()
  return bar() + foo() + baz()
return bar()
```

```
function baz()
  function bar()
  return foo()
return bar()

skip
```

jest poprawny i powinien dać wynik kompilacji podobny do kompilacji poniższego programu.

```
function foo()
return foo_bar()

function foo_bar()
return foo_bar() + foo() + baz()

function baz()
return baz_bar()

function baz_bar()
return foo()

skip
```

Zagnieżdżone funkcje (pełne)

(Opcja `-sublang full`)

Ostatni wariant zadania jest istotnie trudniejszy i wart niewiele punktów. Główną nagrodą za jego rozwiązanie powinna być satysfakcja z rozwiązania nietrywialnego problemu. Teraz pozwólmy funkcjom korzystać ze wszystkich zmiennych widocznych w jej leksykalnym kontekście, tzn. funkcja może korzystać ze swoich zmiennych, zmiennych lokalnych i argumentów wszystkich swoich przodków w hierarchii oraz zmiennych globalnych programu.

To zadanie oczywiście było by łatwe, gdyby maszyna wirtualna wspierała domknięcia, tak jak maszyna wirtualna z drugiej tercji wykładu. Jednak gdy funkcje nie są obywatelami pierwszej kategorii (tak jak w naszym języku), to wystarczy, że funkcja będzie umiała odnaleźć ramkę funkcji[1] rodzica — a wskaźnik na tę ramkę możemy przekazać jako dodatkowy parametr do funkcji. Jak

[1] Fragment stosu na którym znajdują się zmienne lokalne danej funkcji.

odwołać się do zmiennych należących do funkcji-dziadka? To proste! W ramce rodzica odnajdziemy wskaźnik na ramkę dziadka. Dla przykładu rozważmy poniższy program.

```
var g

function foo(x)
  function bar()
  return foo(g+x) + bar()
return 13 + bar()

skip
```

Używając wzorcowego kompilatora, otrzymaliśmy następujący program.

```
     0: ENTER 1
      foo:
     2: ENTER 0
     3: CONST 13
     4: PUSH
     5: TOP
     6: LEA   1
     7: PUSH
     8: CALL  foo:bar
     9: LEAVE 1
    10: PRIM  ADD
    11: LEAVE 0
    12: RET
      foo:bar:
    13: ENTER 0
    14: TOP
    15: LOAD  1
    16: LOAD  2
    17: PUSH
    18: TOP
    19: LOAD  2
    20: LOAD  2
    21: LOAD  0
    22: PUSH
    23: TOP
    24: LOAD  3
    25: LOAD  1
    26: PRIM  ADD
    27: PUSH
```

```
28: CALL  foo
29: LEAVE 2
30: PUSH
31: TOP
32: LOAD  2
33: PUSH
34: CALL  foo:bar
35: LEAVE 1
36: PRIM  ADD
37: LEAVE 0
38: RET
```

Widać tu wszystkie istotne aspekty tego rozwiązania. Dostęp do zmiennej należącej do rodzica wymaga dwóch dereferencji (np. instrukcje 23–25 odczytują wartość zmiennej `x`). Dostęp do zmiennej dalszego przodka wymaga więcej dereferencji (np. instrukcje 18–21 odczytują zmienną `g` z ramki dziadka). Podążanie łańcuchem wskaźników też jest potrzebne do wyliczenia wskaźnika na ramkę rodzica wołanej funkcji (np. instrukcje 14–16 wyliczają adres ramki dziadka funkcji `bar`, który jest rodzicem wołanej funkcji `foo`). Wołając funkcje-dziecko trzeba podać wskaźnik na swoją ramkę, który nie musi być tożsamy z wierzchołkiem stosu — czasami trzeba pominąć wartości tymczasowe na stosie (np. instrukcje 5–6).

Wskazówka

Maszyna wirtualna rozważana w tym zdaniu posiada wysokopoziomowe konstrukcje sterowania, więc zakodowanie przepływu sterowania nie powinno sprawić większych problemów. Główna trudność tego zadania polega na wygenerowaniu tych kawałków kodu, które wykonują dostęp do zmiennych: język źródłowy posługuje się nazwami symbolicznymi, natomiast maszyna wirtualna używa już adresów odnoszących się do stosu. Twoja funkcja kompilująca powinna przyjmować jak parametr środowisko, które opisze co zrobić ze zmiennymi. Środowisko powinno być jakąś abstrakcyjną reprezentacją stosu podczas liczenia danego wyrażenia/instrukcji. W najprostszym rozwiązaniu, do reprezentacji środowiska można użyć typu `var option list`, który opisuje kształt stosu. Wartość `Some` x oznacza, że w tym miejscu na stosie znajduje się zmienna x, zaś `None` oznacza inną wartość (np. wskaźnik powrotu, albo wartość tymczasową).

Testowanie i odrobaczanie

Implementowanie kompilatorów od zarania dziejów było uznawane za trudne zagadnienie. Jedną z przyczyn tej trudności jest dość duża odległość między ujawnieniem się błędu a jego przyczyną. Dlatego swoje rozwiązanie należy dobrze przetestować. W przypadku znalezienia błędu, udostępniony przez nas szablon rozwiązania udostępnia mechanizmy ułatwiające zrozumienie przyczyny błędu.

Po pierwsze, uruchomienie szablonu z flagą `-dump` wyświetla wynik kompilacji kodu. Pamiętaj, że `dune` specjalnie traktuje wszystkie argumenty wiersza poleceń zaczynające się od znaku `-`. Aby przekazać takie parametry do uruchamianego programu, należy je poprzedzić podwójnym znakiem minusa (`--`), np.

```
$ dune exec inf -- -dump -vm no example.imp
```

Drugim ułatwieniem jest to, że szablon udostępnia trzy implementacje maszyny wirtualnej. Opcją `-vm spec` wybieramy maszynę znajdującą się w module `EvalSpec`. Tą maszynę należy uznać za formalną specyfikację. W przypadku wystąpienia błędu, maszyna wybrana opcją `-vm precise` wyświetla trochę więcej informacji na temat napotkanego błędu wykonania. Jest jeszcze trzecia maszyna (`-vm fast`), która akceptuje sporo błędnych programów, ale daje wyobrażenie jak nasza maszyna wirtualna mogła by być wydajnie zaimplementowana. Opcja (`-vm no`) sprawia, że skompilowany program nie jest uruchamiany.

Ponadto, poleceniem `dune utop` możemy wystartować sesję interpretera, `utop` w której wszystkie moduły są dla nas dostępne. W szczególności funkcja `InfLib.parse_file` pozwala wczytać program źródłowy z pliku i przetransformować go do składni abstrakcyjnej.